An automated reasoning system based on isabelle/HOL (Algebras, Languages, Algorithms and Computations)

Kobayashi, Hidetsune; Ono, Yoko

数理解析研究所講究録
第1769巻 2011年 75-82

An automated reasoning system based on isabelle/HOL

Hidetsune Kobayashi
Department of Mathematics
Ninon University

Yoko Ono
Department of Information System
Niigata University of I. I. Studies

1. Introduction

We have some computer languages which can process logical inference and derives some simple propositions equivalent to an originally given proposition. Isabelle, MIZAR, Coq etc. are such proof assistant languages, and some of them have functions which can prove simple propositions automatically. However almost all mathematical propositions are too complicated for the functions to prove completely. And even for a simple proposition a proof with such language needs many tedious logical proof steps, because those languages require strict logical statements having no gap. In human proof, we needn't write down such rigorous logical chain of propositions, because we can easily see almost all such steps are correct or not. Hence human proofs are concise, and they are easier to read than the machine proofs consisting of long rigorous chain of logical expressions.

This observation presents two problems:

1) is it possible to resolve such simple logical steps automatically?

2) how we can give a machine prover a deep mathematical idea to complete a proof ?

This report is an essay on the above problems. Isabelle is developed by Lawrence Paulson and others written in polyML, HOL stands for the higher order logic. We are developing an automated reasoning system called H-prover which works on emacs together with ProofGeneral developed by D. Aspinall which interface with isabelle/HOL.

From now on a proof in isabelle/HOL is called as a "formalized proof". A formalized proof starts with a proposition expressed in logical symbols. Applying axioms, lemmas and theorems already proved, or unfolding definitions, we change the original proposition and obtain some propositions. Repeating such operations step by step, finally when all thus derived propositions are apparently true, then the original proposition is said to be proved. Hereafter, to avoid a confusion, axioms and propositions or theorems already proved are called as "rules", and the word "proposition" means "proposition to be proved". Some proofs in isabelle/HOL is composed by forward inference and others are composed by backward inference. A file containing axioms, definitions, propositions and formalized proofs written in isabelle is called a "theory file" and its file name is given as "XXX.thy", where XXX is a freely given name to express the contents of the file e.g. "Set.thy".

In section 2, we give brief examples of propositions and proofs written in isabelle/HOL to illustrate how a mathematical proposition is expressed and how a proof proceeds. In section 3, we discuss how an automated reasoning system should be designed and we show a scheme of our H-prover. In section 4, we give some examples of commands generated by H-prover semi-automatically, i.e. when H-prover cannot step forward, we can give a command and let the prover generate proof steps as far as possible. We give proofs and try to obtain a feeling of a "mathematical idea". We know almost any mathematical proposition is not proved without some "mathematical ideas", therefore we encounter important problems concerning "mathematical". In section 5, we show more how H-prover works.

2. Proofs in isabelle/HOL

We present two propositions to see how a proposition and a proof is written in isabelle/HOL. The first lemma is very simple and easy to see:

lemma example21:"⟦x ∈ A; A ⊆ B ⟧ ⇒ x ∈ B"

Here the first word lemma is a declaration that the following expression is a proposition to be proved. Example21 is the name of the proposition, after proving the proposition, it is referred to by the name as a rule. In fact, this proposition is already proved as a lemma "subsetD":

lemma subsetD:"⟦c ∈ A; A ⊆ B ⟧ ⇒ c ∈ B"

By using a rule "subsetD", a proof to " Example21" is given as

apply (rule_tac c = "x" and A = "A" and B = "B" in subsetD, assumption+)

By abuse of terminology, we call the above a "command". This command is interpreted as "use the rule subsetD with x instead of c" and then apply assumption+, where + means apply the rule "assumption" as much as possible. ProofGeneral has a function "proof-assert-next-command-interactive" which pass the command to isabelle and isabelle returns propositions after applying the first part of the rule:

1. "⟦x ∈ A; A ⊆ B ⟧ ⇒ A ⊆ B"
2. "⟦x ∈ A; A ⊆ B ⟧ ⇒ x ∈ A"

This is because isabelle applies subsetD as "to see x ∈ B, we have to show two propositions 1 and 2 above. These two derived propositions are called subgoals. In the subgoal 1, we have the conclusion in the assumption, so "apply assumption" resolves the subgoal 1. Similarly, "apply assumption" resolves the subgoal 2. The command "apply (rule_tac c = x and A = A and B = B in subsetD, assumption+)" is concatenated two commands "apply (rule_tac c = x and A = A and B = B in subsetD)" and "apply assumption+". After applying this concatenated command, we have no remaining subgoal, and a proof is finished, we give the command "done", then isabelle is ready to prove next proposition.
Now we give the second lemma:

```
lemma example22:"X ∩ Y = X ⇒ X ⊆ Y"
apply (rule subsetI)
apply (frule_tac P = "λxxx. x ∈ xxx" and s = "X" and t = "X ∩ Y" in ssubst, assumption+)
apply (thin_tac "X ∩ Y = X") (* this line is given manually *)
apply (cut_tac c = "x" and A = "X" and B = "Y" in Int_iff,
drule_tac Q = "x ∈ X ∩ Y" and P = "x ∈ X ∧ x ∈Y" in iffD1,
assumption+, erule conjE, thin_tac "x ∈ X ∩ Y" )
apply assumption+
done
```

Applying the rule subsetI, we have a subgoal

!!x. ⟦X ∩ Y = X; x ∈ X ⟧ ==> x ∈ Y

To this subgoal, we apply the rule ssubst which is a proposition $[\![s = t; P\ t]\!] \Rightarrow P\ s$, and frule_tac inserts the conclusion of the rule applied into premise. In this case, after executing the second command, we have

$$!!x.\ [\![X \cap Y = X; x \in X; x \in X \cap Y]\!] ==> x \in Y$$

This subgoal has a problem that we are falling into an infinite loop, because $x \in X \cap Y$ is rewritten as $x \in X \cap X \cap Y$ by ssubst. A simple way to avoid falling down into an infinite loop is to delete the assumption $X \cap Y = X$ from the premise. So, we put the third command manually. Of course a sophisticated programming can make a system not falling into such an infinite loop. And we note that an application of "thin_tac" is dangerous, because after eliminating it, we cannot use it as an assumption anymore. The fourth command is a composition of commands that separates $x \in X \cap Y$ as $x \in X$; $x \in Y$. After executing the fourth command, we have subgoals having the conclusion appearing already in the premise, hence "apply assumption+" completes the proof. We trace step by step how the expression is rewritten by the fourth command.

cut_tac c = "x" and A = "X" and B = "Y" in Int_iff:

Int_iff is the rule "$(c \in A \cap B) = (c \in A \wedge c \in B)$". We use cut_tac to input the proposition "$(x \in X \cap Y) = (x \in X \wedge x \in Y)$" into the premise of the subgoal as

$$!!x.\ [\![X \cap Y = X; x \in X; x \in X \cap Y; (x \in X \cap Y) = (x \in X \wedge x \in Y)]\!] ==> x \in Y$$

drule_tac Q = "$x \in X \cap Y$" and P = "$x \in X \wedge x \in Y$" in iffD1:

iffD1 is the rule "$[\![Q = P; Q]\!] ==> P$". drule removes the first term in a premise, hence we have

1. $!!x.\ [\![X \cap Y = X; x \in X; x \in X \cap Y]\!] ==> x \in X \cap Y$
2. $!!x.\ [\![X \cap Y = X; x \in X; x \in X \cap Y; x \in X \wedge x \in Y]\!] ==> x \in Y$

assumption+

The subgoal 1 is resolved by this command, while subgoal 2 still remains.

erule conjE

This command separates conjunction, hence we have a subgoal

$$!!x.\ [\![X \cap Y = X; x \in X; x \in X \cap Y; x \in X; x \in Y]\!] ==> x \in Y$$

assumption

Finally "assumption" resolves the last subgoal.

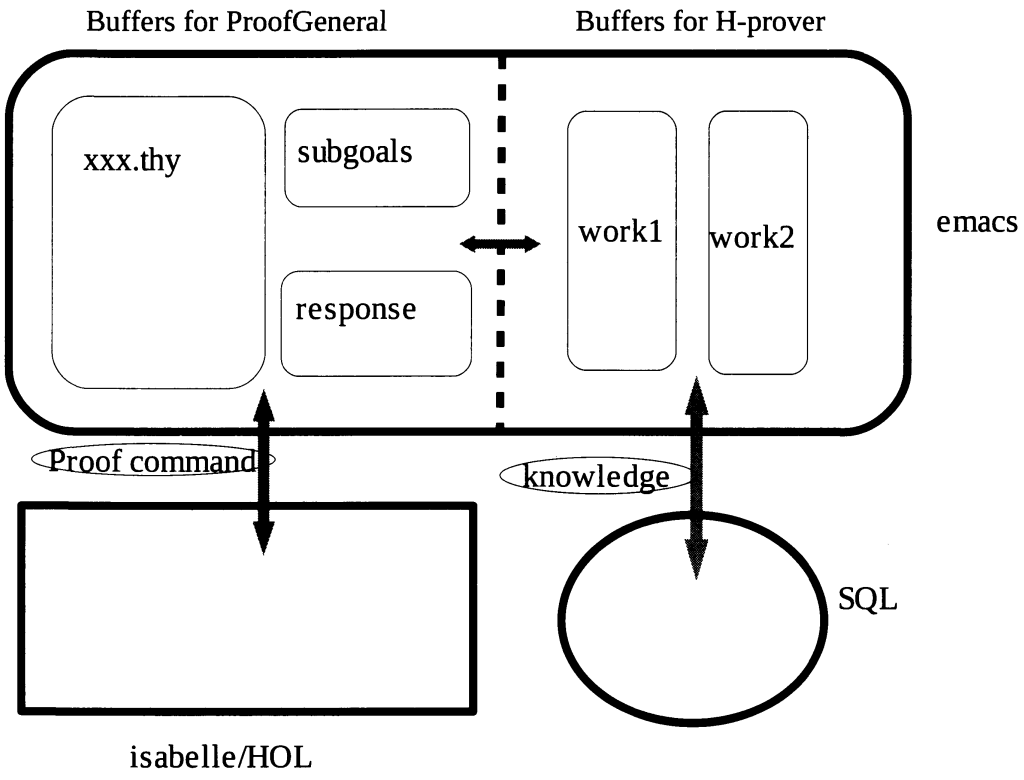
We saw the formalized proof is composed of simple and trivial steps and almost all steps are not worth to read from the mathematical point of view.

3. Design of H-prover

Why we need an automated reasoning system? Is it possible to prove propositions appearing in textbooks of mathematics? Provably nobody can give affirmative answer to these questions. However there are some reasons which encourage us to promote a study of automated reasoning system. At first there is no doubt that formalized proofs scarcely have errors within them. Therefore formalization can perform the role of a proof checker. As we saw in the previous section, a formalized proof consists of many commands only for logical conversion, not representing a meaningful mathematical idea. Here is a task of a prover. We can expect an automated prover to fill up logical gaps between mathematical ideas automatically. We can design a interface of a prover having only mathematically meaningful

statements and completing a proof in the background. Thus we can obtain concise and accurate proofs. This is the first reason why we develop a prover. For a beginner, it is hard to find out a starting point of a proof. There are some systematic study on "how to prove", and we can store in a machine prover the methods of proof. Then an automated prover can be used as a tutor of mathematics. This is the second reason. Finally, we can obtain an answer quickly. From example, if measures to deal with natural calamities are formalized, then we can give the best solution quickly.

Now, we show the scheme of our automated reasoning system "H-prover". It works together with ProofGeneral, so the program is written in emacs lisp. To store the proved propositions, we have a SQL server. Here is the illustration of the scheme of H-prover.



In the illustration, the horizontal arrow implies communication of subgoals from the left to the right and solution from the right to the left. In an ordinary isabelle session, the buffer xxx.thy is set to be a current buffer. Here, xxx.thy is the name of a file containing axioms, definitions and propositions written in isabelle language. The buffer "subgoal" is used to show subgoal(s) given by isabelle after reading a proposition or command into. "response" is a used to show some notification generated by isabelle. The buffers "work1" and "work2" are used to get statements from/to isabelle and SQL to/from H-prover.

H-prover has two modes for generating commands. One is step by step mode, that is press specific keys and H-prover executes a command and generates a next command. Another mode is full automatic mode, that is press specific keys and H-prover tries to generate next commands as much as possible. In both two modes, when H-prover cannot generate next command, it stops and wait for human assistance. So, we input a command and press specific keys, then it begins two work. Thus the prover composes a proof partly automatically and partly asking human assistance.

4. H-prover with human assistance

We show how H-prover works by giving an example. The first simple example is to show a proof given by human assistance. It should be noted that the proposition in the first example is obvious, and human proof can be written in one line, but a mechanical proof requires some lines:

Lemmas and commands	isabelle's response
lemma A2set_A1:" ⟦ A1 ⊆ A; f ∈A→A1 ⟧ ⇒ A2set f A A1 ⊆ A1"	⟦A1 ⊆ A; f ∈A→A1⟧ ⇒ A2set f A A1 ⊆ A1

H-prover apply a general rule to prove a set is a subset of another.

apply (rule subsetI)	!!x. ⟦ A1 ⊆ A; f : A → A1; x ∈ A2set f A A1 ⟧ ⇒ x ∈ A1
This is a general method to check subset.	
apply (simp only:A2set_def)	!!x. ⟦ A1 ⊆ A; f : A → A1; x ∈{x ∈ A1. ∃y∈A - A1. ∃n. itr n f y = x} ⟧ ⇒ x ∈ A1
Expand the definition of A2set, manual.	
apply (erule CollectE)	!!x. ⟦ A1 ⊆ A; f : A → A1; x ∈ A1 ∧ (∃y∈A - A1. ∃n. itr n f y = x) ⟧ ⇒ x ∈ A1
H-prover doesn't know CollectE, given manually.	

CollectE is a proposition:

[| a ∈ {x. P x}; P a ==> PROP W |] ==> PROP W

Since a ∈ {x. P x} is in the premise, we have only to show P a ==> PROP W, this is rewritten as the last subgoal rewritten properly by isabelle/HOL. Finally the command

```
apply (rule_tac P = "x ∈ A1" and Q =
      "∃y ∈ A - A1. ∃n. itr n f y = x"
        in conjunct1)
```

gives a trivial proposition, and assumption is the final command to complete the proof. Thus when the prover does not know how to treat a set of the form {....}, it cannot proceed anymore. If we give some proper commands manually, then H-prover begins to work.

There are two cases human assistance is required. The case that problems can be resolved by programming and another case is that problems are not resolved by programming only. In the latter case we need human idea which is called "mathematical idea" hereafter in this report. To obtain feeling of mathematical idea, we give the following proposition.

Lemma. Let T be a subset of S, and let (S, ≤) and (T, $\lhd$) be well ordered sets, where $\lhd$ is the restriction of ≤ to T. If ψ is an order isomorphism of S to T. Then for all x in S, we have x ≤ ψ(x).

Proof. Suppose there is an x0 in T such that ψ(x0) < x0. The set M = {x ∈ S. ψ(x) < x} is non-empty. Since M is non-empty, it has the minimum element m. ← **Mathematical idea "make the set M and consider the minimum element m".**

We have an inequality $\psi(m) < m$. Since ψ is an order isomorphism, we have $\psi(\psi(m)) < \psi(m)$ ←
Mathematical idea "apply ψ to the inequality $\psi(m) < m$".
This shows that $\psi(m)$ belongs to the set M and $\psi(m) < m$, this contradicts to the fact m is minimal.

We present one more example that a proof is relatively long but it is generated completely. A line beginning from the middle is an isabelle response.

lemma may26_2:"⟦X ⊆ S ; Y ⊆ S ; X ⊆ Y ⟧ ⇒ (S - Y) ⊆ (S – X)"
⟦X ⊆ S; Y ⊆ S; X <= Y ⟧ ⇒ S - Y ⊆ S – X

apply (rule subsetI)
!!x. ⟦X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y⟧ ⇒ x ∈ S – X

apply (subst Diff_iff)
!!x. ⟦X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y⟧ ⇒ x ∈ S & x ∉ X

apply (rule conjI)
1. !!x. ⟦ X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y ⟧ ⇒ x ∈ S
2. !!x. ⟦ X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y ⟧ ⇒ x ∉ X

apply (rule_tac c = "x" and A = "S" and B = "Y" in DiffD1, assumption)
!!x. ⟦X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y ⟧ ⇒ x ∉ X

apply (rule_tac P = "x ∈ X" in notI)
!!x. ⟦X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y; x ∈ X⟧ ⇒ False

apply (frule_tac c = "x" and A = "X" and B = "S" in subsetD, assumption+)
!!x. ⟦ X ⊆ S; Y ⊆ S; X <= Y; x ∈ S - Y; x ∈ X; x ∈ S ⟧ ⇒ False

apply (frule_tac c = "x" and A = "X" and B = "Y" in subsetD, assumption+)
!!x. ⟦X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y; x ∈ X; x ∈ S; x ∈ Y ⟧ ⇒ False

apply (rule_tac c = "x" and A = "S" and B = "Y" in DiffD2)
1. !!x. ⟦X ⊆ S; Y ⊆ S; X ⊆ Y; x ∈ S - Y; x ∈ X; x ∈ S; x ∈ Y⟧ ⇒ x ∈ S - Y
2. !!x. ⟦X ⊆ S; Y ⊆ S; X ⊆Y; x :∈ S - Y; x ∈ X; x ∈ S; x ∈ Y⟧ ⇒ x ∈ Y

apply assumption+
No Subgoals

It is seen that each step is only a simple logical manipulation.

5. How H-prover generates a proof

At first, we present some types of expressions which can be resolved into simpler expressions by some commonly used method. For H-prover, those expressions are easy to treat. The last example in the previous section shows that resolution for those expressions are classified into two classes. One is mathematical elementary resolution and another is logically elementary resolution. Since it is not proper to list up all those resolutions here, we present only some simple types of logical expressions and resolution to those expressions.

splitting a conclusion:

lemma: ⟦ P1; P2; P3; ... ; Pn ⟧ ⇒ Q1 ∧ Q2 ⟦P1; P2; P3; ... ; Pn ⟧ ⇒ Q1 ∧ Q2

apply (rule conjI) $[\![P1; P2; P3; ... ; Pn]\!] \Rightarrow Q1$

$[\![P1; P2; P3; ... ; Pn]\!] \Rightarrow Q2$

splitting a conjunction in premise:

lemma: $[\![P1 \wedge P2; P3; ... ; Pn]\!] \Rightarrow Q$ $[\![P1 \wedge P2; P3; ... ; Pn]\!] \Rightarrow Q$

apply (erule conjE) $[\![P1; P2; P3; ... ; Pn]\!] \Rightarrow Q$

take an arbitrary element:

lemma :P $\Rightarrow \forall x.\ Q\ x$ $P \Rightarrow \forall x.\ Q\ x$

apply (rule allI) $!!x.\ P \Rightarrow Q\ x$

The last subgoal is taken as a proposition P $\Rightarrow$ Q x except one point that x is a bounded variable. In the proposition P $\Rightarrow$ Q x (not following to !!x.), x is taken as a fixed constant. In a forward inference, sometimes we need more expression derived from premise.

Adding an expression to premise:

lemma: $[\![P1; P2; P3; ... ; Pn]\!] \Rightarrow Q$ $[\![P1; P2; P3; ... ; Pn]\!] \Rightarrow Q$

Suppose we have a rule aaa: $[\![P1; P2]\!] \Rightarrow R$,

apply (frule aaa)

inputs R into premise of the lemma as $[\![P1; P2; P3; ... ; Pn; R]\!] \Rightarrow Q$

Proving a proposition having the conclusion $\neg$ P, H-prover employs the rule notI.

Prove not Q:

lemma: $[\![P1; P2; P3; ... ; Pn]\!] \Rightarrow \neg Q$ $[\![P1; P2; P3; ... ; Pn]\!] \Rightarrow \neg Q$

apply (rule notI) $[\![P1; P2; P3; ... ; Pn; Q]\!] \Rightarrow$ False

In addition to those logically simple expressions as above, there are some mathematical elementary expression that H-prover can resolve automatically. Since there are too much to list up, we present only one such elementary expression. We note that those mathematical elementary properties are give by L. Paulson and others in Set.thy included in Isabelle2009.tar.gz.

lemma: $[\![Z \subseteq X ; Z \subseteq Y]\!] \Rightarrow Z \subseteq X \cap Y$ $[\![Z \subseteq X ; Z \subseteq Y]\!] \Rightarrow Z \subseteq X \cap Y$

apply (rule_tac A = "X" and B = "Y" and C = "Z" in Int_greatest, assumption+)

In the above, a combination of Int_greatest and assumption+ resolves the lemma. H-prover, storing rules in a data base, checks whether a rule within a database is applicable or not after reading into it a proposition to be proved. Even an elementary mathematical property to show that a set is a subset of another set, there are several methods according to premise of the proposition. "subsetI" is one of them, but this is used commonly, and since this is a powerful method to show a set is a subset of another set, we put it as a last resort of H-prover to show that.

Within H-prover, we transform an expression into a tree as

$\exists c.\ P\ c \wedge Q\ c \quad \leftrightarrow \quad$ (ex-S c d-S and-S (P c) (Q c))

A variable in a tree is specified by its position from the root. For example, the location of the variable c following P is child child child left-child child (abbreviated as ccclce), were e means "take the root". Therefore, even a variable is appearing in multiple places, we can specify the location of it. In the above tree, ex-S stands for $\exists$, d-S stands for the period and and-S stands for $\wedge$. Using tree, we can treat a variable and a function in the same way:

$$!!x.\ [\![X \cap Y = X; x \in X]\!] \Rightarrow x \in Y$$

apply (drule_tac P = "λxxx. x ∈ xxx" and s = "X" and t = "X ∩ Y" in ssubst)

1. !!x. x ∈ X ==> x ∈ X
2. !!x. ⟦x ∈ X; x ∈ X ∩ Y⟧ ⇒ x ∈ Y In this

case difference between x and Y is only a left-child and a right-child, because the conclusion x ∈ Y of the first subgoal is expressed in tree as (in-S (x) (Y)). We may say that tree expression is appropriate to handle HOL (higher order logic).

6. Proof direction, future work

A proof is a well arranged sequence of propositions each derived from the former. In general, there are several propositions derived from one proposition, therefore if a selection of some proposition(s) at each step is not appropriate, then the sequence does not reach the goal. We may call "a proof direction" choice of one proposition from the propositions derived from the original one. Therefore proof directions give a sequence of propositions, and some of them are proofs. H-prover, inputting a proposition, generates a next possible commands and store them as a list. Then H-prover sends a command to isabelle from the list, and isabelle, after inferring, returns the result as a subgoal. Repeatedly, H-prover tries to step forward from the given subgoal.

We have no evaluation which command is likely to give a good direction. Therefore, at present, we have no standard to control a proof direction. So we have big problem for an automated reasoning system. One is how we can take mathematical idea into the system and another is how to control proof directions. One method to make the control easy is adopt parallel computation, that we test the possible commands parallelly and choose the sequence reaching the goal. If the branch of the search path is not so much, then this method will work. However, in general, the number of branches is not so small to get a sequence reaching the goal.

References

[1] Bourbaki, N. Élèments de Mathématique, Théorie des ensembles, Springer.
[2] Troelstra, A. S. and Schwichtenberg, H., Basic Proof Theory, Cambridge tracts in theoretical computer science 43, 1996.
[3] Velleman, D. J., How to prove it, Cambridge University Press, 1995.
[4] Paulson, L. et al., A proof assistant for higher order logic, Lecture notes in computer science, Springer, 2002.
[5] Kobayashi, H. and Ono, Y., On an automated reasoning system "H-prover" , Dec. ,2010, CACS2010,CD-R, IRAST.
[6] Kobayashi, H and Ono, Y. An Application of the Formal Methods to Statistics, Proceedings of the 2009 International Symposium on Computing, Communication and Control, pp238-241.
[7] Kobayashi, H. and Ono, Y., Type and cardinality in Isabelle/HOL, Proceedings of the 10-th symposium on algebra, language and computation, pp.1-4. 2008.